箱詰めパズルのプログラムについて (計算機によるパズル・ゲームの研究)

AUTHOR(S):

川合, 慧

CITATION:

川合, 慧. 箱詰めパズルのプログラムについて (計算機によるパズル・ゲームの研究). 数理解析研究所講究録 1976, 263: 62-74

ISSUE DATE:

1976-02

URL:

http://hdl.handle.net/2433/105823

数理解析研究所講究録
263巻 1976年 62-74

62

箱詰めパズルのプログラムについて

東大 理学部　川合 慧

1. はじめに

箱詰めパズル（packing puzzle）とは，一定の形状をした箱（board）に，それをちょうど埋めつくすだけの面積（あるいは体積）の和を持つ小片（駒，piece）の集合を，互に重ならないように収容するという問題である。この種の組合せ問題を計算機を使用して解く場合には，賢明な人間が行なうようなしらみつぶし法を，計算機の速度に物を言わせて実行するのが一般的である。その際，箱の中で各pieceが単独にとることのできる位置の集合をまず求め，その直積集合を作り，そのすべての要素（pieceの配置法）についてpieceの重なりがあるかどうかを調べることによって，この問題は原理的に解くことができる。ところが，本文で述べるTetraballパズル（piece数11）に例をとってみても，この直積集合の要素数は約10^{22}と非常に大きい。一年が約3×10^{13}μ秒

しかないことから考えてもわかるように，このようなしらみつぶし法は，数学的（原理的）解法ではあっても実際的な解法とはなり得ない。そこで，逐次的なはめこみを行ない，失敗したら一段階戻るという，backtrackingの手法を用いることになる。これは，pieceの重なり状態に関する何らかの強力な定理が発見されない限り，解を見出す，もしくは数え上げるための唯一の方法である。

Tetraballパズルについては[1]において，パズルの記述と数え上げプログラムの実際，および効率についてのいくつかの考察を発表したが，今回は，探索プログラムの振舞といくつかの実験結果，および対称解の除去方法について述べる。

2. Tetraballのプログラム

Tetraballの解の数え上げプログラムについてまず簡単に述べる。プログラムとパズルの詳細については[1]を参照のこと。

(i) Tetraballパズル。　各pieceは単位球を4個，tetrahexおよびtetrominoの形式で平面状につないだもので，11種類ある。boardは単位球44個を正八面体状に組上げた形。

(ii) Pieceのデータ。　各pieceについて，それが位置し得

る場所（site）の表を作る。各siteはpieceの要素(unit)†が位置する場所（cell）†の番号の組で表わされる。

(iii) 探索順序。 Backtrackingを行なう場合，各段階における目標は，一定の規準により決定される一つの空cell（target-cell）を埋めることである。未使用pieceのsiteで，このtarget cellを含むものがすべて試される。target cellの決定法としては，最も回りを囲まれているcell，すなわち，最も最小路に近いcellをとる方法（Most Closely Surrounded Cell：MCSC†法）と，最も番号の若いcellをとる方法（Youngest Empty Cell：YEC†法）の2つを採用した。

(iv) 実行結果。 TOSBAC-40（加算時間5.6 μ秒）において，MCSC法で120時間，YEC法で27時間を費して数え上げを行なった。両方法とも，解の総数として同じ数，7,482が得られた。これにより，解の信頼性はかなり高いものと思ってよいであろう。

3. 探索プログラムの振舞

上で述べたように，MCSC法とYEC法では総探索時間に大きな開きがある。この原因を明らかにする目的で，探索プロ

† [1] では箱をframe，pieceの要素もelement，frameの要素もboxと呼んだが，以後これも board，unit および cell と呼ぶことにする。

プログラムが実際にどのように動いているかを計測した。その方法の概略を下に示す。このプログラムは，いくつかのpieceがすでにはめこまれている状態をパラメータとして与えられ，それを含むすべての解を数え上げる。

```
proc search(n); integer n;
  begin determine target cell;
        for all unused piece do
          for all site of piece
              whichincludes target cell do
            begin D1[n] := D1[n] + 1;
                  if locatable
                    then begin D2[n] := D2[n] + 1;
                               locate;
                               search(n+1);
                               remove
                         end
            end
  end
```

D1 はこの手続きの呼び出し一回につき試されるsiteの数，D2 は同じく成功したsiteの数を数える。もちろん，最終結果はこれらの総和を表わすことになる。計測した結果は表1の通りである。また，その大体の様子を図1に示す。

まず第一に気付くことは，D1 のグラフの形状が，MCSC法とYEC法でほとんど同じだということである。また，総探索回数もほぼ等しい（YEC法の方が約4%だけ多い）。このことから，単位操作の回数に関して見る限り，この2方

表-1　探索回数

n	D1	D2	D1	D2
2	(MCSC) 372	182	(YEC) 372	286
3	15,043	4,423	24,253	10,152
4	337,772	65,134	585,310	193,778
5	4,485,241	579,577	8,597,038	1,891,952
6	33,782,113	3,004,889	62,084,829	10,450,954
7	142,719,435	8,480,673	204,212,363	29,253,669
8	310,114,084	11,653,097	352,338,894	33,291,452
9	296,389,355	6,392,729	249,284,038	17,580,934
10	96,855,877	979,897	50,815,744	2,226,013
11	6,150,874	7,482	1,762,973	7,482
SUM.	890,850,166		929,705,814	

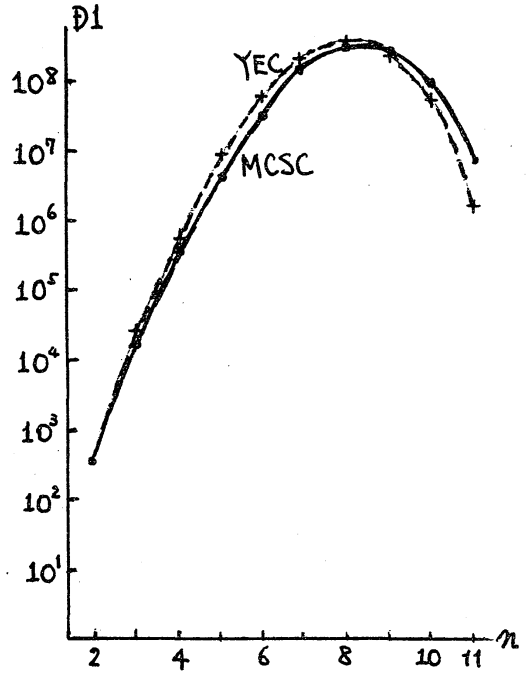

図1.　試行site数

法はほぼ同等であるということが言える。したがって，4倍以上という計算時間の差は，MCSCを決定するための手間に帰せられる。pieceを置くのに必要な操作は，YEC法ではsiteで示される4つのcellにpiece名を書き込むことだけであるが，MCSC法ではこれに加えて，そのまわりの空cellの「囲まれ数」を増加させる操作が必要となるからである。MCSC法は「制限

の強い場所から試す」という，backtrackingにおける効率化条件を満すべく採用したが，本パズルでは，ほぼ同じ効率を持つYEC法と較べてoverheadが大きすぎたようである。

つぎに特徴的なこととして，探索過程が圧倒的にn=7,8,9の所に集まっていることがあげられる。実際，D1[1]+…+D1[6]はMCSC法では全体の4.3%，YEC法では7.7%にすぎないのに対し，D1[7]+D1[8]+D1[9]は84.1%（MCSC）及び86.7%（YEC）を占めている。探索回数に関するこのグラフの形状については，データが少いので確定的なことは言えないが，一般的にほぼ似たような状況にあると思われる。これは，探索効率を高めるための枝刈り操作を導入するに当って，考えに入れておかねばならない事実である。すなわち，出現頻度の小さい段階（tetraballではpiece数が6以下の段階）で精力的に枝刈りを行なえば，その手間が少々大きいものであっても，全体に対する寄与が大となる可能性が大きいのである。MCSC法では，「囲まれ数」操作の副産物として、pieceを置いたことによって生ずる孤立した空cellの存在を検知することができる。実際のプログラムではこれを枝刈り用に用いているが，上述と同様の計測を行なった結果，これが有効となるのはpiece数が8以上の段階であることがわかった。したがって，このcheckが実行時間の短縮にほとん

ど寄与していないのは明らかである。

前回，[1]において探索効率の見積りを行なったが，そこでは次の表式を導いた。

$$\alpha_n = S\left(1-\frac{n}{N}\right) \qquad \cdots\cdots (*)$$

$$\beta_n = S\left(1-\frac{n}{N}\right)^M \qquad \cdots\cdots (**)$$

ここで α_n, β_n は piece が n個おかれている一つの配置に対して，試される site の数と成功した site の数，S は piece 当りの平均 site 数，N は piece 数，M は piece の大きさである。ここでは，すでに置かれているn個の piece と，これから試す piece の各 unit が，board 中に一様に存在するものという，きわめて粗い近似を用いている。この α_n と β_n に対応するものとして，つぎの量を計算した。

(i) D1[n]／D2[n-1]

これは，平均的には α_n に一致する量と考えられる。MCSC法の場合にはかなりきれいな直線となっており，α_n の式の形とよく合っている。YECについては，一様分布近似が成り立たないことがわかる。このグラフ

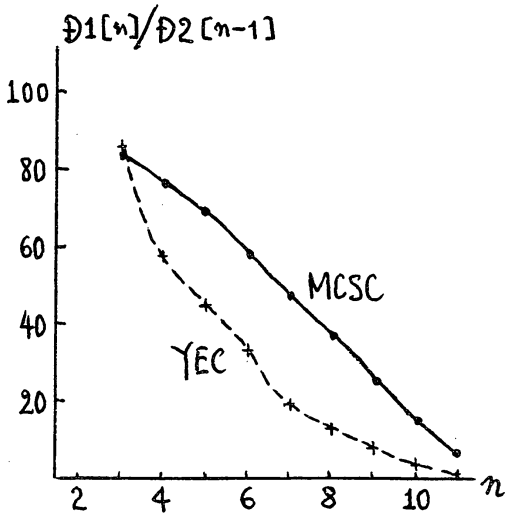

図2. 一段あたりの平均試行 site 数.

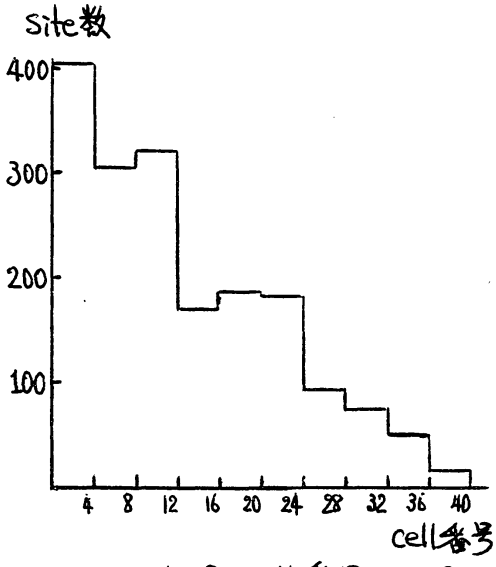

図3. 先頭cell番号による site数の分布

の形は，図3に示す先頭cell番号による site数分布の形によって説明される。YEC法において，置かれている piece数とYECの番号は必ずしも比例するわけではないが，おおよその形状は一致するものと思われる。

(ii) $[D2[n]/D1[n]]^{\frac{1}{3}}$

これは前述の $((**) \div (*))^{\frac{1}{3}}$ に対応し，上の表式では $1-\frac{n}{N}$ に対応する。直線的な部分が長いので，非常に荒い近似では (*), (**) が成立しているとも言えるが，(i)で述べたようにYECでは (*) が成立していないので，この図からだけでは結論はでない。

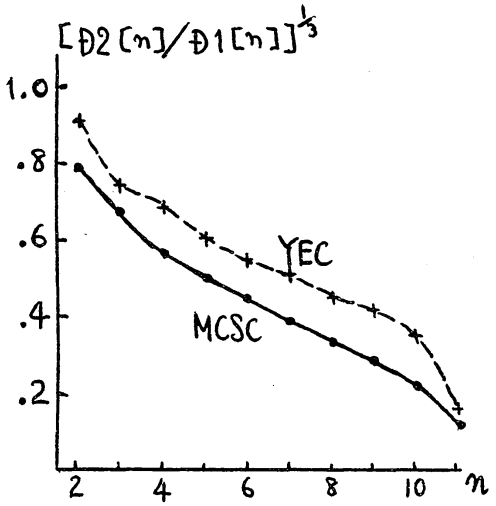

図4. 一段あたりの平均試行成功率
(各unitについて)

図1は，本プログラムが扱った search tree の各レベルの node 数と見ることができるが，D.E. Knuth が[2]において，この形状および全体の node 数の見積り法について述べている。

4. 対称解の取り扱いについて

箱詰めパズルでは，回転や反転でお互に移り変り得る解は，まとめて1個と数えられるのが普通である。これを取り扱うには，特定のpieceの置き場所を制限するのが一般的なやり方であるが，場合によっては複数個のpieceを考慮に入れる必要が生ずることもある。ここでは，対称解の間の変換が，boardをそれ自身に移す対称変換に限られることに着目した取扱いを述べる。

4-1 siteと配置に対する変換

まず，boardの対称変換群を

$$G = \{g_0, g_1, \cdots, g_n\} \qquad \cdots\cdots(1)$$

とする。g_0は恒等変換である。各g_iはcellに対する置換の一つである。つぎに，piece P の j番目のsiteを

$$S_j^P = \{C_{j1}^P, \cdots, C_{jM}^P\} \qquad \cdots\cdots(2)$$

で表す。Cはcellの番号，Mはpieceの大きさである。また

$$S_0^P = \{0, 0, \cdots, 0\} \qquad \cdots\cdots(3)$$

をpiece Pが未使用であることを示すsiteとして導入しておく。これにしたがって，Gの要素g_iは$\{0\}$を$\{0\}$に，$\{1, \cdots, V\}$を$\{1, \cdots, V\}$に移すものと拡張しておく。siteに対する変換を，要素cellに対する変換の結果をまとめたM-組であると定義すれば，つぎの性質が存在する。

$$\forall g \in G,\ \forall p\ \forall j : \exists k,\ g S_j^p = S_k^p \qquad \text{-----(4)}$$

任意の piece配置は，つぎのように表わされる。

$$T = (S_{j_1}^1, S_{j_2}^2, \cdots, S_{j_N}^N) \qquad \text{-----(5)}$$

N は piece数である。つぎに，site と配置を不変に保つ変換の集合を定義する。

$$h(S_j^p) = \{ g \in G \mid g S_j^p = S_j^p \} \qquad \text{-----(6)}$$

$$H(T) = \{ g \in G \mid gT = T \} \qquad \text{-----(7)}$$

ただし，配置に対する変換は，各 site を変換したもののN-組である。$T_0 = (S_0^1, \cdots, S_0^N)$, $T = (S_{j_1}^1, \cdots, S_{j_N}^N)$ とすると

$$H(T_0) = G \qquad \text{-----(8)}$$

$$H(T) = \bigcap_{p=1}^{N} h(S_{j_p}^p) \qquad \text{-----(9)}$$

が成り立つ。

4-2 対称解の除去

今，2つの解 R_1 と R_2 ($R_1 \neq R_2$) が

$$R_2 = g R_1$$

なる関係にあったとする。数え上げ過程でどちらか1つだけを取扱うためには，両者を区別するものが必要であるが，それは明らかに

$$g \notin h(S_j^p)$$

を満たす R_1 中の site である（一般に複数個存在する）。

backtrackingで T_0 から R_1 を得るには，T_0 中の各 S_0^p を（ある順番で）S_j^p $(j\neq 0)$ に置き換えてゆくわけであるが，

$$g \in H(T_0),\quad g \notin H(R_1)$$

であるから，はじめて対称性 g をこわす site を扱った時に，それに g をほどこした site に印をつけ，それに関する探索は禁止するようにすれば，対称解 R_2 は数えられることがない。これが対称解除去の原理である。

4-3 実際の操作法

Tetraball パズルでは G の要素は24個あるが，"J" と呼ばれる piece のすべての site について

$$h(S) = \{g_0\}$$

が成り立っている。したがって "J" の site（144個）を G で分割したものの代表（6個）のみを扱うことによって，対称解を完全に除くことができる。

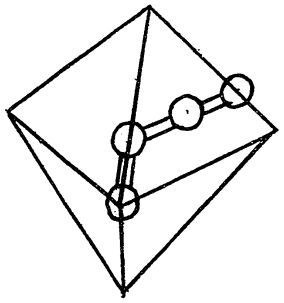

図5. Tetraballの board と piece "J"

しかし，問題によってはこのような性質を持つ piece が存在しないこともある。実際，pentocube パズルにおいて，4個の piece を置いても対称性がくづれない例が [1] に報告されている。

対称解の除去をプログラムで行なう方法としては，各site間の G による関係をあらかじめ全部計算しておく方法と，

backtracking を進めながら実行時に処理する方法とが考えうれるが，実行時に行なう方がプログラムが明解であり，3. で述べた考察によりそれほど効率は落ちないと予測される。概略のプログラムを下に示す。

```
proc prelocating(piece); integer piece;
  for all site of piece do
    if not mark[piece, site] and locatable
      then begin locate;
                 marking; change symmetry;
                 if yet symmetric
                   then prelocating(piece + 1);
                   else search;
                 recover symmetry;
                 remove
           end
```

marking は，site S_{site}^{piece} に G の要素を作用させ，結果の site に mark を付ける。主プログラムは prelocating (1) を呼ぶ。prelocating は，対称性がなくなるまで自分自身を呼び続け，それから search を呼ぶ。対称性が残っている間は試行 piece の順番は固定されている。

この方式のプログラム実験はまだやっていないが，この方法が確立すれば，プログラミングの中核は

(i) piece と変換 G の表現

(ii) site 表の作成

の2点になり，速度の記録を競う場合以外の標準的手法とな

ろう。

5. まとめ

当初，"絶体的高能率アルゴリズム" であると思って採用したMCSC法が，さほどの成果を上げなかったこと，および，1974年3月の研究会で通研の竹内氏が話された「10億回」という数字に興味を持ったことから，総試行回数を計ってみた。その結果は以上に述べた通りであるが，その他に，竹内氏のプログラム[1]が扱った探索treeとTetraballのそれとが，ほぼ同じサイズであることも判明した（表1参照）。このあたりが，計算機で数時間程度で数え上げが可能な問題の標準的大きさであろう。これから考えて，計算機による数え上げが（現実問題として）可能な最大のサイズは約1兆（$=10^{12}$）程度であろうと思われる。

参考文献

[1] "計算機によるゲーム・パズルの具体化の検討," 京都大学数理解析研・講究録 217 (1974)

[2] D.E. Knuth, "Estimating the Efficiency of Backtrack Programs", Mathematics of Computation, vol 29, no 129, Jan. (1975) pp. 121-136.